METHOD AND APPARATUS FOR VERIFYING INVARIANT PROPERTIES OF DATA STRUCTURES AT RUN-TIME

Inventor

Carol L. Thompson
6937 Calabazas Creek Circle
San Jose, CA 95129

Jeff Littfin
424 26th Avenue
San Mateo, CA 94403

Assignee

Hewlett Packard Company

METHOD AND APPARATUS FOR VERIFYING INVARIANT PROPERTIES OF DATA STRUCTURES AT RUN-TIME

FIELD OF THE INVENTION

The present invention generally relates to run-time verification of program correctness, and more particularly to compiler-based support for run-time verification of invariant properties of data structures.

BACKGROUND

The values assigned to the various data structures defined in a computer program are often intended to posses invariant properties. For example, a particular variable may be limited to a certain range of values. The invariant property of the variable is that $l <= x <= u$, where l is the lower bound of the range, u is the upper bound of the range, and x is the value of the variable. Another example is a variable that is constrained to be an aligned pointer to a 32-bit value, i.e. its low-order two bits must be zero.

During program execution, a data structure is updated one or more times with values that depend on the application logic and data input to the application. To protect against events during program execution that may result in data corruption or a fatal program error, application developers often program defensively to avoid these scenarios. For example, input data are checked for acceptable ranges.

While defensive programming catches many error conditions before they can cause serious problems, there may be some events that the developer has not considered. For example, an error in application logic may result in a data structure being assigned a value

that is outside the invariant range. To create source code that checks every update to every data structure would unreasonably bloat the source code and make the source code difficult to read. Furthermore, while application developers are often aware of the invariant properties of data structures, compiler and program analysis tools cannot deduce these invariant properties. Thus, the task of guarding against violation of invariant properties is left to the developer.

A method an apparatus that address the aforementioned problems, as well as other related problems, are therefore desirable.

SUMMARY OF THE INVENTION

The invention provides a method and apparatus for verifying invariant properties of data structures of a computer program during program execution. Code that verifies whether a runtime value of the data structure is consistent with the invariant property is automatically generated from an annotation of the data structure in the source code. In executing the program, the runtime value of the data structure is compared to the invariant property in the automatically generated code. If the runtime property is inconsistent with the invariant property, the program branches to exception handler code.

Various example embodiments of the invention are set forth in the Detailed Description and Claims which follow.

BRIEF DESCRIPTION OF THE DRAWINGS

Various aspects and advantages of the invention will become apparent upon review of the following detailed description and upon reference to the drawings in which:

FIG. 1 is a flow diagram that illustrates an example transformation of source code to executable code in accordance with one example embodiment of the invention; and

FIG. 2 is a flowchart of a process for verifying invariant properties of data structures at run-time in accordance with one embodiment of the invention.

DETAILED DESCRIPTION

The present invention is believed to be applicable to a variety of programming languages and has been found to be particularly useful for compiled programming languages. While the invention is not limited to compiled languages, various aspects of the invention will be appreciated through a discussion of an example embodiment involving compiled languages. Those skilled in the art will appreciate that the invention is also applicable to interpreted languages.

According to one aspect of the invention, selected data structures in the application source code are annotated with respective invariant properties. The term "data structure" refers to both data structures that are language provided (e.g., integers, real numbers, character strings, etc.) and data structures that are user defined and application specific. For each annotation, code is automatically generated to enforce the invariant property, and the code is associated with the data structure. Exception handler code is also automatically generated to handle violations of the invariant property. During execution of the application, the run-time values of the data structures are checked against the invariant properties, and the exception handler code is invoked if an invariant property is violated.

FIG. 1 is a flow diagram that illustrates an example transformation of source code to executable code in accordance with one example embodiment of the invention. Block

102 represents the source code of an application, block 104 represents the symbol table generated in compiling the source code, and block 106 represents the executable code generated during compilation.

Source code 102 includes an example declaration 108 of a data structure, T, having a field, F. An invariant property of the data structure is annotated at reference 110 using the C or C++ language *#pragma* statement. For field *F* of the data structure *T*, the invariant property is a range of data values having a lower bound of *lower_boundF* and an upper bound of *upper_boundF*. At reference 112 in the source code, field F of the data structure T is updated.

In a first phase of compiling source code 102, symbol table 104 is generated. Along with the information that is conventionally stored in the symbol table, the invariant properties of data structures as defined in the source code 102 are included. For example, at reference 116, the description of the invariant property for field F of data structure T is included in the symbol table entry. In addition to the description of the invariant property, in one embodiment the code location(s) of the update(s) to the data structure is stored in the symbol table. In an alternative embodiment, the code locations are identified at code generation time based on references to the symbol table for data structures having the invariant property descriptions.

The code generation phase of compilation uses the information from the symbol table 104 to generate executable code 106. At reference 120 in the executable code, field F of data structure T is updated. Following the code that updates the data structure, code is included at reference 122 to check whether the invariant property has been violated. The code to check the invariant property is automatically generated. Included in the code

is a conditional branch to the exception handler code at reference 124. The exception handler code is also automatically generated from the invariant annotation.

In addition to the invariant property that is a range of data values, other types of invariant properties include a range of data addresses and a range code addresses. For an invariant property that is a range of data addresses, the generated code checks whether the value in the annotated data structure falls within the valid or specified range of data addresses. Similarly, for an invariant property that is a range of code addresses, the generated code checks whether the value in the annotated data structure falls within the valid or specified range of code addresses. In one embodiment, the developer need not specify the range of valid addresses for code or data address ranges; rather, the compiler determines the valid range of addresses for code or data.

FIG. 2 is a flowchart of a process for verifying invariant properties of data structures at run-time in accordance with one embodiment of the invention. At step 202, selected data structures in the source code program are annotated with respective invariant properties according to program requirements. In one embodiment, the developer specifies the invariant properties in the source code with language-specific statements. The properties may be specified with a conventional source code editor or a source code editor with language-specific capabilities.

In one example embodiment, invariant property annotations are specified in the C or C++ language with a pragma statement of the form:

#pragma *property_type [l..u]* T <F>

where *property_type* is one of data_address, code_address, or data_range; *l* and *u* are the lower and upper bounds of the range; T references a data structure; and <F> is optional depending on whether the data structure T has any fields. The referenced data

structure may reference an object of language-provided data type (e.g., integer or character string), an instance of an application-defined type, or all instances of an application-defined type. The code and data address ranges are optional.

At step 204, the source code is compiled, and at step 206 the invariant properties of the data structures are recorded in the symbol table. In addition to storing the invariant properties, at step 208 the initial compilation phase stores in the symbol table the code locations where annotated data structures are updated. The code locations inform the code generator where invariant checks need to be inserted in the executable code. In another embodiment, only the invariant properties are communicated through the symbol table, and the code generator determines where invariant checking needs to be performed based upon statements that update to the annotated data structures.

In yet another embodiment, the annotations of invariant properties are associated with individual variables or objects rather than with a particular data type or class. In this embodiment, the code that checks the invariant properties only checks individual instances rather than all instances of a particular type or class.

At step 210, code is generated to test whether the run-time properties of the data structures satisfy the invariant properties. Each update to an annotated data structure is followed by a check of the run-time property. The code checks whether the value of the data structure lies within the specified range of values. If the range check fails, the action taken depends upon the language. For non-object-oriented languages, the action is to abort the program. For object-oriented languages, the action raises an exception that provides the user with the option to handle the error.

Example pseudocode that describes the code generated to verify the run-time properties is:

```
condition1 = ( a < LowerBoundA )
branch on condition1 to AssertLabelA
condition2 = ( a > UpperBoundA )
branch on condition2 to AssertLabelA
```

Example pseudocode that describes the exception handler for non-object-oriented code is:

```
AssertLabelA:
print: . Variable a outside expected range of type A at
file foo.c, line 5
print:  Assertion declared in file foo.h, line 20
```

Example pseudocode that describes the exception handler for object-oriented code is:

```
AssertLabelA:
create an AssertionException object, with the variable
name and source location of the update, along with the
information about the assertion (e.g. class name and
where the assertion was declared), and throw that
exception.
```

At step 212, the program is executed. If during program execution the run-time property of a data structure does not comply with the invariant property, the program is directed to the exception handler code, as shown by step 214.

In another embodiment, the number of segments of code that are generated to check invariant properties can be controlled either by the compiler or by a developer-selected threshold. These techniques are described following patents/applications, which assigned to the assignee of the present invention and hereby incorporated by reference:

"METHOD AND APPARATUS FOR PERFORMING CORRECTNESS CHECKS OPPORTUNISTICALLY" by Carol L. Thompson, filed on November 21, 2000 and having patent/application number 09/717,570.

"METHOD AND APPARATUS FOR VARYING THE LEVEL OF CORRECTNESS CHECKS EXECUTED WHEN PERFORMING CORRECTNESS CHECKS OPPORTUNISTICALLY USING SPARE INSTRUCTION SLOTS" by Carol L. Thompson, filed on November 21, 2000 and having patent/application number 09/718,059.

The present invention is believed to be applicable to a variety of programming languages and has been found to be particularly applicable and beneficial in compiled languages, for example, C and C++. Other aspects and embodiments of the present invention will be apparent to those skilled in the art from consideration of the specification and practice of the invention disclosed herein. It is intended that the specification and illustrated embodiments be considered as examples only, with a true scope and spirit of the invention being indicated by the following claims.